Brief Overview of the Portable Document Format (PDF) and Some Challenges for Text Extraction

Tim Allison, NASA's Jet Propulsion Laboratory, California Institute of Technology
January 12, 2022

Disclaimer

"The research was carried out at the NASA (National Aeronautics and Space Administration) Jet Propulsion Laboratory, California Institute of Technology under a contract with the Defense Advanced Research Projects Agency (DARPA) SafeDocs program.

The author would like to thank Peter Wyatt, Chief Technology Officer of the PDF Association, and other colleagues for their feedback on this article. All errors and omissions are the author's.

The following represents the viewpoints of the author and does not represent the funding agencies or reviewers."

Introduction

The Portable Document Format (PDF) is one of the most common document file formats used in industry, academia and government. PDF comprises a significant component of files on the internet (see "PDF's Popularity Online"). For non-technical users, PDF files may seem straightforward and largely reliable. However, in practice, PDF files present a rich set of challenges for tools that extract text to enable search or other natural language processing tasks. The goal of this article is to offer a general overview of some of the challenges in extracting text from PDFs for technically-oriented people who may be new to PDF. Specifically, this paper is intended for those who process PDF "in the wild", which is to say, developers or development teams which do not have control over the generation of the PDFs they are processing. For those who are able to influence how the PDFs they process are generated, we encourage focusing on the final section of this article.

Text extraction is a critical precursor component of information retrieval systems (see "Reliable Electronic Text"). When text extraction fails, users cannot find the documents or content they need. The text extraction component of search systems is often overlooked; we hope to use this article to improve the state of the practice for information retrieval engineers' handling of PDF files.

We use the terms "PDF writer" and "PDF reader" to distinguish software packages that generate PDFs from those that render or otherwise extract information from PDFs. Obviously, many applications do both, such as tools that allow editing of PDFs.

In the following, we use "text" to refer to text that a sighted human would recognize as "primary page text," including headers, footers, footnotes and so on. There are other pieces of text that may also be stored in PDFs that may or may not be rendered on the page or extracted by PDF readers, including, among others, annotations, bookmarks, optional content, alternative text and layer names. Given the scope of this article, we focus on extracting the primary page text.

We begin with a brief history of PDF, offer a brief overview of PDF file structure and then offer a deep dive of how text may be stored in PDFs. We then turn to potential challenges with extracting text from PDFs before ending on a note of hope for ongoing improvements for the format.

Brief History

The PDF format was developed by Adobe Inc. in the 1990s (see: "History of PDF" and "Who Created PDF"). The primary goal was to enable a consistent presentation of a document across platforms, operating systems and applications. The format was based on the Postscript language, which was initially developed to define page elements for driving laser printers. It took a number of years for the PDF format to take off (see: "Why the PDF is Secretly the World's Most Important File Format"). In 2007, the PDF format was released to the

International Organization for Standardization (ISO). The first ISO specification was published in 2008 (ISO 32000-1:2008) and an extensive update for PDF 2.0 was published in in 2017 (ISO 32000-2:2017) with a revision in 2020 (ISO 32000-2:2020).

While Adobe has remained a key source for PDF writers and readers (with the Adobe Acrobat and Acrobat Reader applications), numerous other commercial and open-source projects have also developed writers and readers. All PDF writer tools vary in their adherence to the specification, making it challenging to ensure that different PDF readers will interpret the same information in the same way.

Mainstream tools for verifying adherence to the specification include Adobe's Preflight tool and veraPDF which focuses specifically on PDF/A, an ISO-standardized restricted subset of PDF designed for archiving and long-term preservation.

Recently, Peter Wyatt has led a revolutionary project – the Arlington PDF Model – to create an open access, vendor neutral, specification-derived machine-readable definition of all formally defined PDF objects and their intra- and inter-object relationships (see: Wyatt_LangSec21.pdf). This project has already identified numerous areas for improvement in Adobe's Dictionary Validation Agent (DVA), used in a component of Preflight. More broadly, though, the Arlington PDF Model will have a transformative effect on the reliability of PDF writers and readers (and PDF files!) for decades to come because it is so comprehensive and openly and freely available.

Basics of PDF Structure

In the following, we offer a basic overview of the underlying structure of a PDF. We encourage the reader to review the specifications for more details, as this account necessarily glosses over many details.

In the following, we've used Microsoft Word to create a document that contains a single page with the text "Hello World!", and we've saved that as a PDF.

A PDF file is composed of a header, a body (list of objects), a cross reference (xref) table and a trailer as shown in Figure 1.

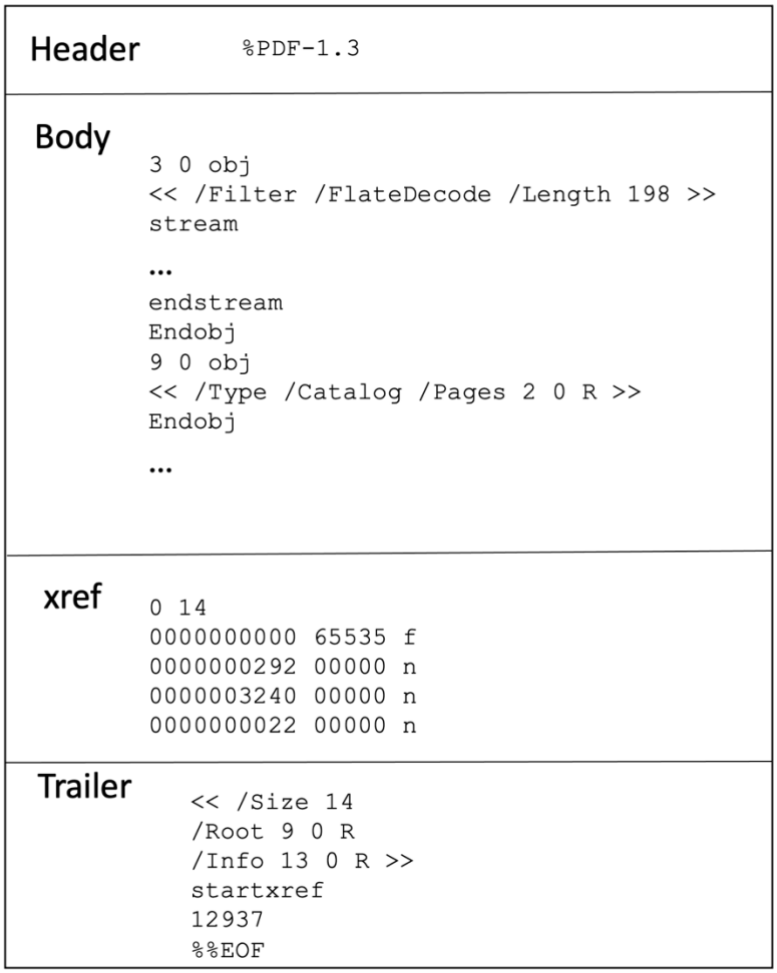


Figure 1: Example PDF File Structure

Critical parts of a PDF file are stored at the end of the file, and PDF readers typically read the trailer and the cross reference (xref) table before deeply processing the body contents. The trailer contains the byte offset to the active xref table (byte 12937 in Figure 1) and an indirect reference to the document's root ("9 0 R" in Figure 1 – PDF object references take the form of "ObjectNumber GenerationNumber 'R'", so object number 9 with generation 0).

From these components, PDF readers construct a directed graph of the document structure; this graph may include cycles. In Figure 2, we show a high-level overview going from the Root/Catalog object down to the Page element. If there's more than one page, the Pages object (2 0 R) will contain an array of Page references or other Pages references.

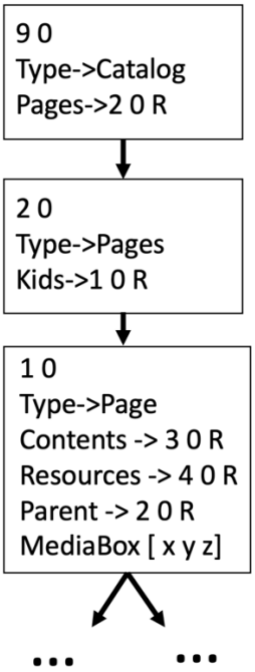


Figure 2: Document Structure from Catalog

Text in PDFs

PDFs store three primary types of data that need to be considered for text extraction: images, vector graphics and electronic text. Images include standard image formats – JPEG, among others – and, these images may be modified as they are rendered by operators in the PDF file, such as scaling, rotation, color manipulation, or transparencies (on transparency in PDF, see: "20 Years of Transparency in PDF"). Images may contain text, as when a paper document is scanned, and the images of the pages are stored in the PDF. Vector graphics, which are composed of drawing commands, may be used to draw text on a PDF page (see Figure 3, a vector graphics-based PDF created by Tilman Hausherr for TIKA-3270. Finally, PDFs may, of course, contain electronic text. To render text, PDFs often store embedded font files and mappings between values in content streams and Unicode character values.

Note that a given PDF page may contain any combination of the three elements: images, vector graphics and electronic text. Further, "text" may be represented by all three elements. Optical character recognition (OCR) is the only option to generate electronic text from images and vector graphics. Extracting electronic text as text should be trivial. However, after a brief example, we'll turn to catalog some of the ways that extracting electronic text may be challenging.

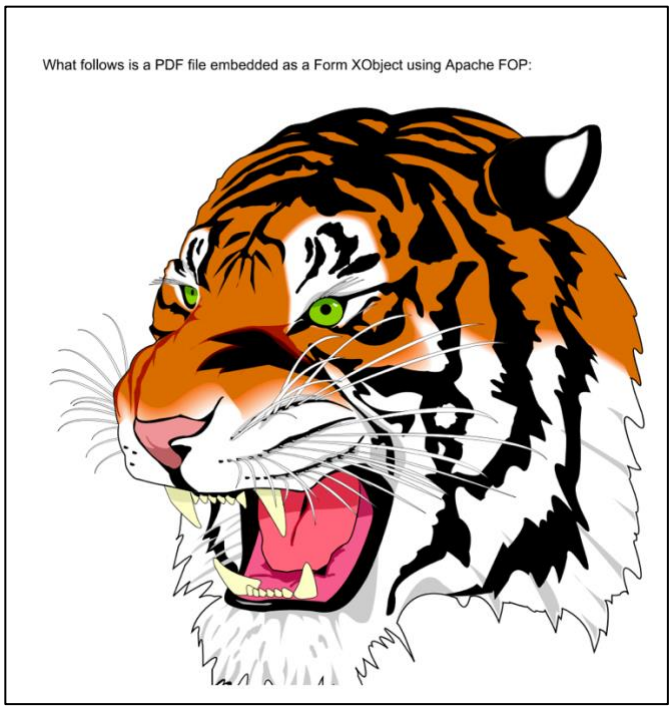


Figure 3: Example of Vector Graphics

Extracting text: A Simple Example

In Figure 4, we use Apache PDFBox's PDFDebugger to show the internal structure of the page object for page 1 of the file we've been using as an example. This page contains a content stream ("Contents") and resources, including font information and an embedded TrueType font file. Content streams in PDFs are typically compressed through one or more compression algorithms.

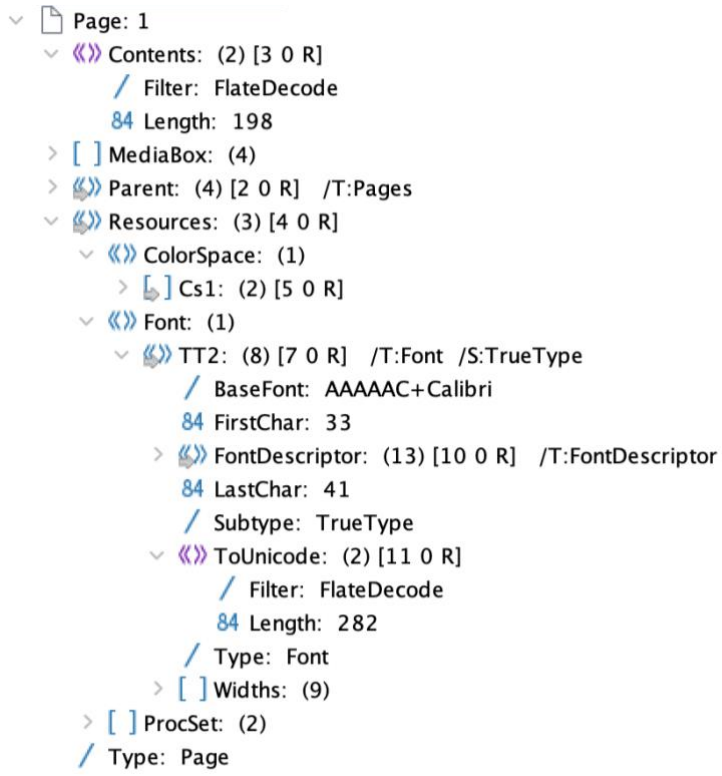


Figure 4: Document Structure, Page 1

This content stream is compressed with a FlateDecoder. The raw bytes of the stream as stored in the file are shown in Figure 5 and the uncompressed stream is shown in Figure 6.

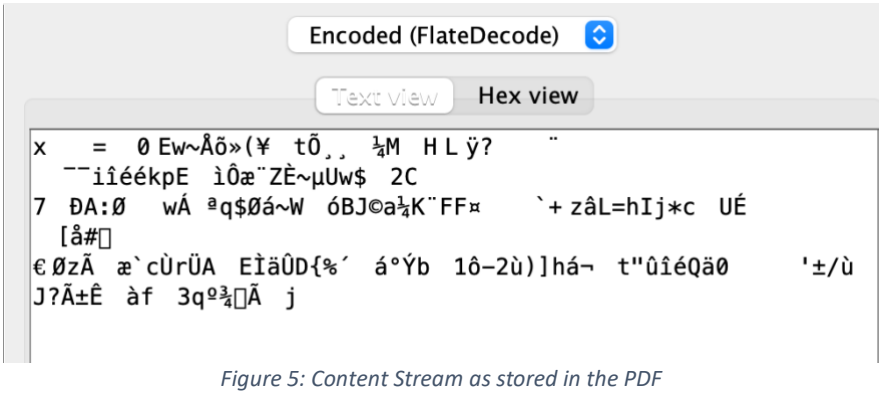


Figure 5: Content Stream as stored in the PDF

The content stream (Figure 6) contains the stream operators that draw the text on the screen. As an example of how to write the 'H' in "Hello," the run starts with 'BT' (begin text), the location and scaling/rotation are set via the text matrix ('tm' operator), the font (named TT2 in this case) is selected with the 'Tf' operator, the character code '!' is written to the screen with the 'Tj' operator, and then the run is ended with 'ET' (end text).

```
q
  /Cs1 cs
  0 0 0 sc
  q
    0.24 0 0 0.24 8.3906 593.7694 cm
    BT
      50 0 0 50 267.0003 476 Tm
      /TT2 1 Tf
      (!) Tj
    ET
  Q
  q
    0.24 0 0 0.24 8.3906 593.7694 cm
    BT
      -4.0E-4 Tc
      50 0 0 50 298.1527 476 Tm
      /TT2 1 Tf
      [ ("#) -1 (#) -1 ($) ] TJ
    ET
  Q
```

Figure 6: Page 1's Content stream decompressed

Astute readers will have observed that the character code '!' is not the letter 'H'. Given that PDF's goal is for precise appearance, there is no inherent need to store actual Unicode characters here, and historically, character codes (or glyph indices) are sufficient to point to the glyph that is to be rendered. To recover the Unicode value for the character code, the page's resources contain a `ToUnicode` element for this font that maps the characters in the stream to Unicode characters (Figure 7). Specifically, immediately below 'beginbfrange', the table states that hexadecimal value 21 (decimal 33, character '!') should be mapped to hexadecimal value 48 (decimal 72, character 'H').

In the next 'TJ' operator (in Figure 6), note the '-1' values between the character codes; these effectively specify x-coordinate adjustments before writing the next character. We leave it as an exercise for the reader to do similar mapping with an ASCII chart to extract the "ello" from the second text run in Figure 6.

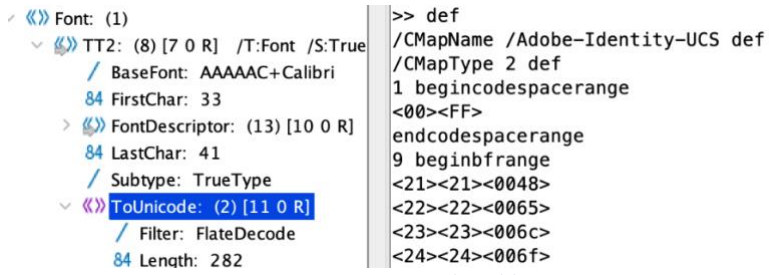


Figure 7: ToUnicode Table

Different PDF readers offer different levels of access into the underlying structure of the PDF. Apache PDFBox, for example, groups the two BT->ET runs into a single string, which can be accessed programmatically via `writeString()`. The call to `writeString()`, includes the text string as well as a list of "text positions", basically, page coordinates for each character,

along with information about direction, scale, rotation and other features for each character. In Figure 8, we've computed the minimum and maximum x and y coordinates for this string.

```
page=1 text="Hello World! " minX=72.47 maxX=134.62 minY=83.99 maxY=91.57
```

Figure 8: PDFBox's writeString()

With this, we wrap up this simple example of how electronic text may be stored in a PDF. The key point is that this is a simple example for "Hello" and glosses over a vast array of complexities of storing and extracting text from a PDF. In the following section, we catalog some of the more common challenges.

Extracting Electronic Text

Extracting electronic text from PDFs is a notorious challenge for those who have worked closely with the file format. In 2006, Michael Kay wrote the following on the challenges of extracting text and structure from PDFs ("How we can convert pdf data into xml?" Figure 9):

> Converting PDF to XML is a bit like converting hamburgers into cows. You may be best off printing it and then scanning the result through a decent OCR package.

Figure 9: Michael Kay on Converting PDF to XML in 2006

In the first subsection, we'll cover some of the challenges with low level text extraction, and in the next section we'll cover some of the higher-level text extraction challenges such as document segmenting and structural tagging. We encourage the reader to see Bogdan's article for FilingDB which identifies similar challenges.

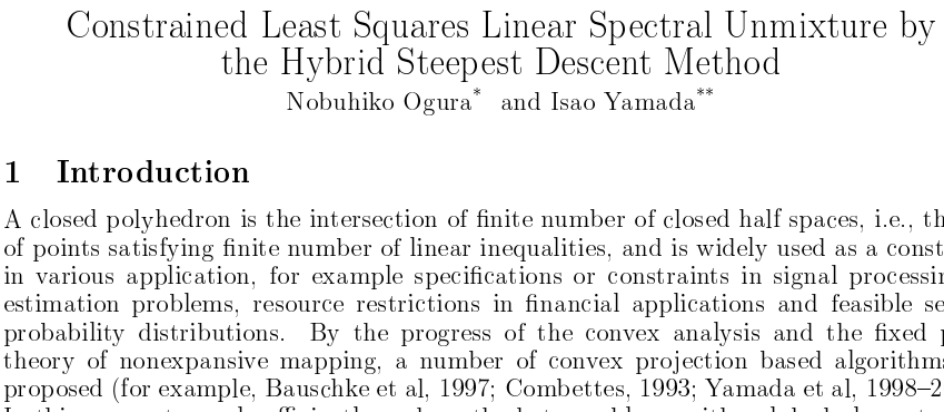
Extracting Electronic Text – Some Basic Challenges

First, people generating PDFs may choose to forbid text extraction, either from the entire document or from portions. PDFs may be encrypted or require a password to decrypt the text. If the PDF reader software doesn't have the password, no text will be extracted. Further, even when unencrypted, PDFs may contain a permission that forbids extracting text. The text is still available, and an ill-configured or malicious PDF reader may extract the text, but it is against the wishes of the people or organization who created the PDF, and most PDF readers will respect this permission. A technique to prevent the extraction of portions of text is redaction, where authors or editors of PDFs will use advanced PDF tools to prevent the rendering and the storage of underlying text from specific portions of a file.

We turn now to more text specific challenges. Mapping the character in the content stream to a Unicode character can be a challenge if the PDF and/or the embedded font lacks a `ToUnicode` mapping, or if the font is not embedded or is corrupt. PDF defines 14 standard Type 1 fonts that do not need to be embedded, but many people want to use fonts beyond those. One of the solutions for ensuring that files are rendered the same across platforms was

to embed fonts outside of the 14 standard fonts inside the PDF file. The drawback is that some font files can be quite large, and if a PDF file contains many fonts, most of it would be taken up with embedded font files. Some PDF writers optimize fonts by storing partial font files that contain information only for the characters/glyphs used in the PDF file. Other writers simply don't store the font files and "hope" that the recipient will have the font on their system. Depending on the font type, some font files may include mappings from characters to Unicode characters. Further, font widths and heights are critical for reconstructing spaces and new lines as we discuss below. When PDF readers have problems with embedded fonts, they typically fallback to fonts stored on the machine on which it is operating; this can lead to different text extracted based on different font information on different operating systems or individual machines.

To give a concrete example of missing Unicode mappings, let's consider 2002_Ogura_1_web.pdf. To a sighted human, this is easily readable (Figure 10).



Constrained Least Squares Linear Spectral Unmixture by the Hybrid Steepest Descent Method

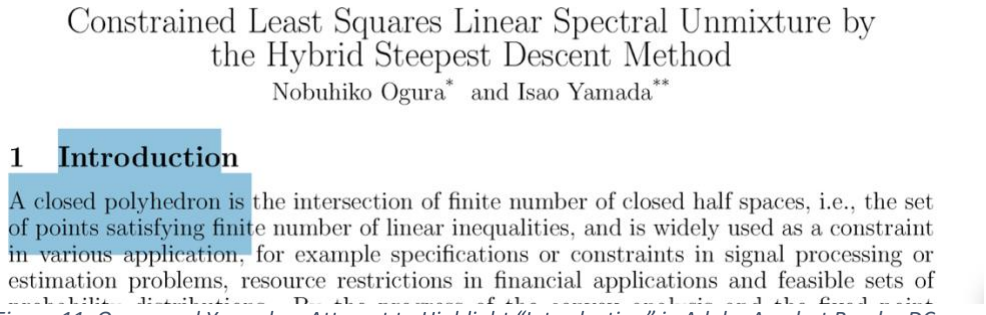
Nobuhiko Ogura* and Isao Yamada**

1 Introduction

A closed polyhedron is the intersection of finite number of closed half spaces, i.e., the set of points satisfying finite number of linear inequalities, and is widely used as a constraint in various application, for example specifications or constraints in signal processing or estimation problems, resource restrictions in financial applications and feasible sets of probability distributions. By the progress of the convex analysis and the fixed point theory of nonexpansive mapping, a number of convex projection based algorithms are proposed (for example, Bauschke et al, 1997; Combettes, 1993; Yamada et al, 1998–2002).

Figure 10: Ogura and Yamada. "Constrained Least Squares Linear Unmixture by the Hybrid Steepest Descent Method." Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) Workshop Proceedings. 2002.

The first sign to a human that something is amiss with the underlying text, though, comes when attempting to highlight "Introduction," and the cursor doesn't correctly capture the desired content (Figure 11). This is with Adobe Acrobat Reader DC.



Constrained Least Squares Linear Spectral Unmixture by the Hybrid Steepest Descent Method

Nobuhiko Ogura* and Isao Yamada**

1 Introduction

A closed polyhedron is the intersection of finite number of closed half spaces, i.e., the set of points satisfying finite number of linear inequalities, and is widely used as a constraint in various application, for example specifications or constraints in signal processing or estimation problems, resource restrictions in financial applications and feasible sets of

Figure 11: Ogura and Yamada – Attempt to Highlight "Introduction" in Adobe Acrobat Reader DC

When the user copies and pastes that highlighted section into a text file, the trouble becomes clearer (Figure 12).

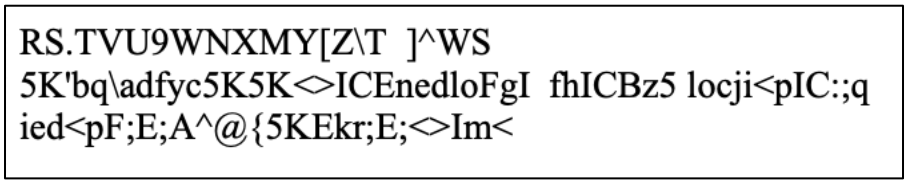


Figure 12: Ogura and Yamada – Text Copied and Pasted for "Introduction"

When we run pdftotext (a command-line tool used commonly in information retrieval systems to extract text from PDFs), we receive text of similar quality as when we manually copied and pasted the text from Adobe Acrobat Reader DC. Note that the text that would be processed by a search system or any other natural language processing system is completely useless (Figure 13).

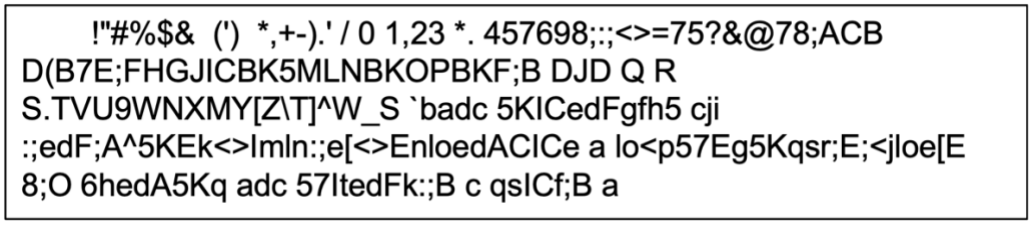


Figure 13: Ogura and Yamada – Text Extracted by pdftotext

When we run Tesseract, an open-source OCR engine, on the file, we get slightly noisy but far more reasonable text (Figure 14).

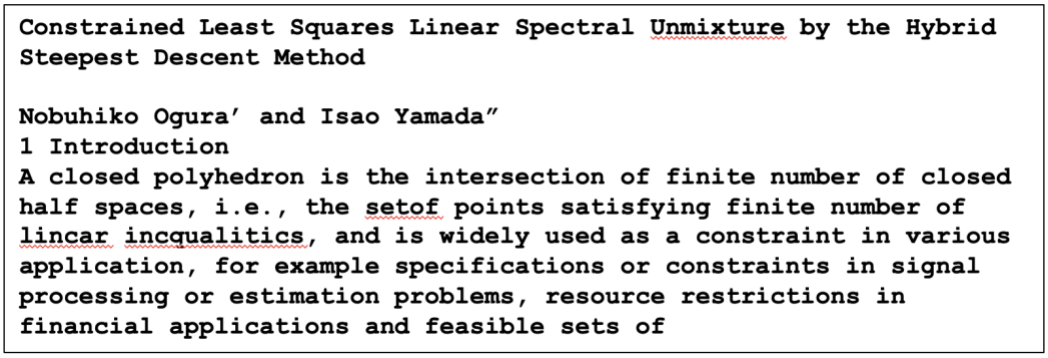


Figure 14: Ogura and Yamada – Text from Tesseract-OCR

In short, missing `ToUnicode` mappings, and missing or corrupted font files can wreak havoc on text extraction.

As we showed above, text is stored as a run with a coordinate on the page as a start point and then character codes. There is no logical connection between the runs, and there is no requirement that a single run contain a meaningful unit: word, sentence, paragraph or that text is rendered in reading order. This has several consequences.

Some PDF writers do not store space characters in the text. A PDF may encode a run “hello” and then effectively move the cursor the distance of a white space and then draw a run for “world”. In this case, the PDF reader needs to understand character widths for the given font and determine when to inject a space so that the extracted text is “hello world” and not “helloworld”. PDF readers may make mistakes when calculating when spaces should be inserted and insert them where they don’t belong, as in, “H ello world”, for example.

More broadly, because the commands to render the text on the page may be stored in any order, PDF readers typically extract all runs from a page and then sort them by location on the page to reconstruct the likely reading order. A sighted human looking at the rendered page will see no problems and will be able to interpret the reading order effortlessly. A PDF reader, however, must do its best to reconstruct the proverbial cow from the hamburger, and try to bring together the text runs into a natural reading order and inject new lines and spaces that may not be encoded in the text runs.

In addition, there may be operators on text runs to rotate them or print them backwards, which makes the reading order even more complicated to reconstruct. These are some of the challenges with reconstructing the text from what is stored in the PDF that affect all languages. There are further challenges for right-to-left languages (Arabic, for example) and for languages that are sometimes written vertically (Japanese, for example). Some resources for some of these challenges include: Alexey Subach’s talk and Antenna House, Inc.’s post.

Another source of surprise for people extracting text from PDFs is that text can be hidden. A sighted human would not be able to see some text and may be surprised to have “extra text.” Text may be hidden in numerous ways. Some common ways of hiding text (intentionally or accidentally) include: text may be written off the page, it may be the same color as the background, it may be too small to read, it may overlap with other text or it may be covered up by an image.

Finally, electronic text may not be “born digital.” When paper documents are scanned, some scanners will run OCR and store the page as an image as well as the OCR’d text as electronic text. This OCRed text uses PDF’s special “invisible text” render mode and is placed on top of the scanned image so that users have the perception of text selection. If the scanner was older, the image quality poor, or there was a suboptimal OCR engine, the quality of this electronic text may be less than ideal.

Extracting Electronic Text – Document Segmentation/Tagging Issues

Going back to the quote about converting PDFs to XML, part of the point is that there should be some logical structure extracted from PDFs. Because of the way that text is often stored, many PDFs contain no information about the reading order (as discussed above) nor about the logical elements such as headers, footers, tables, tables of contents, footnotes or endnotes.

Text or numbers in what sighted readers would effortlessly recognize as tables may be extracted out of order or be concatenated together.

"Two column" PDFs may have text extracted completely out of reading order, with text from one line in the left column followed by text on the same line in the right column.

Extracted text may include header and footer text injected into the middle of a logical paragraph that was split over a page break. In Figure 15, we show a footer in 582116main_GRAIL_launch_press_kit.pdf and the text that was extracted from that file. For applications that expect logical sentences or paragraphs, this kind of word soup can be extremely problematic.

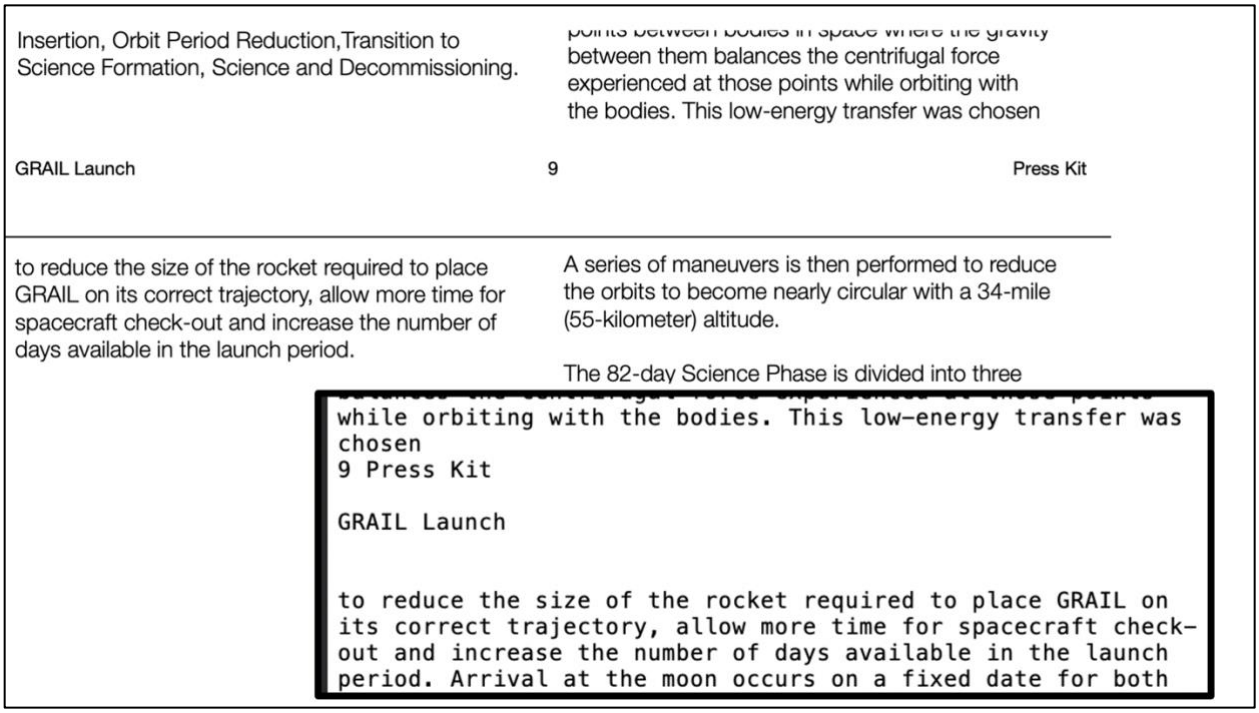


Figure 15: Example of Footer Text

To get a sense of the extent of this problem and the difficulty in fixing it, in mid-2021, Adobe launched a PDF extraction API that uses Liquid Mode, a machine-learning based approach, to automatically segment untagged PDFs. In his article on this launch ("Adobe Launches PDF Extraction Generation APIs") Don Fluckinger quotes Vibhor Kapoor, senior director of marketing for Adobe Document Cloud:

> Recognizing and tagging different elements of a PDF for the purposes of automation has been a vexing technical problem over the years for both Adobe and third-party vendors of PDF tools, said Vibhor Kapoor, senior director of marketing for Adobe Document Cloud.

Figure 16: Kapoor on Adobe's Liquid Mode

That Adobe would identify this as a "vexing technical problem" and yet only offer an API to attempt to handle this in mid-2021 should cause great pause for any development team setting out to solve their own "PDF problem."

There are entire academic journals and conferences that focus on improving the state of the art in extracting structural information from documents generally, and these communities include important work on PDFs, specifically. Some of the more important venues include: the International Conference on Document Analysis and Recognition (ICDAR), International Journal on Document Analysis and Recognition (IJDAR) and Association for Computing Machinery (ACM)'s Symposium on Document Engineering (DocEng). There are also robust research communities for image processing, which can be applied to images in PDFs or page images as stored in PDFs. See for example the Computer Vision Foundation and the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

Further, there are numerous projects that have attempted this kind of automated tagging on specific genres of PDF files, with varying degrees of success. For example, GROBID is designed to extract information from academic papers. Peter Murray-Rust has championed extracting not just structure from untagged PDF, but also processing scientific figures to extract data from images in academic journal articles. For an overview of his Content Mine, see PeterMurrayRust.pdf and for an example of extracting data from vector images see: arxiv.org 1709.02261 and for an overview of his repositories github petermr).

There are also projects to extract specific types of tags/structured elements such as tables, for example, TabulaPDF in the Java ecosystem and Camelot in the Python ecosystem.

In short, extracting structural information from untagged, heterogenous PDFs continues to be, to quote Kapoor, a vexing problem.

Not All is Lost, Reasons for Hope

Before going over some mitigations and potential reasons for optimism, I must acknowledge broad swaths of the PDF file format relevant to text extraction that this brief account has, necessarily, not covered.

We have not gone into detail on the numerous PDF varieties with their own standards, such as PDF/A (Archive) and its variants, PDF/UA (Universal Accessibility), PDF/E (Engineering) and others.

PDFs may contain all types of embedded files, including "standard attachments" (another PDF or a PowerPoint file, which, in turn, may have its own attachments), JavaScript, Extensible Metadata Platform (XMP) metadata, Extensible Forms Architecture (XFA) forms, native PDF form data, multimedia files, color profiles, font files, thumbnail images, electronic signatures, and many others. Information retrieval systems will likely want to process some of these resources and not others.

PDFs may include incremental updates, which means that earlier versions of the document may be trivially reconstructed by truncating the file. Some use cases for text extraction might require recovering the earlier versions of the PDF.

This account has not covered malicious or intentional attacks on text extraction such as Markwood et al's. "PDF Mirage: Content Masking Attack Against Information-Based Online Services" or Chen et al.'s "Attacking Optical Character Recognition (OCR) Systems with Adversarial Watermarks."

As noted earlier, PDFs may contain images that will yield reasonable text when processed with OCR. Image extraction offers a similar set of challenges as text. Briefly, extracting images requires applying transparencies or any other modifications to the images that are stored in the PDF such as cropping, rotation, resizing. As with text runs, images may not be stored as logical images, but a single logical image may be stored as hundreds of smaller images that, when rendered, look like a single image to a sighted human. However, these picture pieces, unless they are rendered together as a single image, are useless for OCR or other types of image processing.

Despite all of the challenges mentioned above, there are some tools and mitigations that can be applied, and there are some reasons for hope.

First, for tools, Johan van der Knijff recently wrote a nearly encyclopedic post on open-source tools for PDF processing and analysis: "PDF Processing and Analysis with Open Source Tools". There are numerous commercial tools, as well, that will inspect and repair PDFs.

As for mitigations, a combination of text extraction and OCR is typically required to handle a broad range of heterogeneous PDFs. Some signs that OCR may be indicated include:

1) Fonts with missing Unicode mappings
2) Few extracted electronic characters (may be an image-only PDF?)
3) Junk text

The first two are straightforward. There is no perfect solution for identifying junk text as in (Figure 13). However, Ashok Popat's "A panlingual anomalous text detector" lays out some

methods for this kind of detection. In a similar vein, the Apache Tika project has developed a module (tika-eval), that computes the out-of-vocabulary statistic for text that is extracted from a file. The module runs language identification on the extracted text and then counts how many of those words are in the lookup list of the top 20,000 words for that language. The module has lookup lists for nearly 130 languages. If the extracted text has a high out-of-vocabulary percentage, OCR may improve the extraction (see also: "Methods for Evaluating Text Extraction Toolkits").

Now, reasons for hope. The PDF Association and stakeholders around the world have collaborated to develop two standards that dramatically improve the reliability of the stored text. The first is the PDF/A standard, originally released in 2005, and its many conformance levels which help improve the reliability of archived PDFs (see PDF/A and PDFA-in-a-Nutshell_1b.pdf). The second is the PDF/UA (PDF/Universal Accessibility) standard, which offers methods for encoding reading order and tagging structural content (headers, footers, tables, etc.) within PDFs. First and foremost, PDF/UA is intended to improve accessibility and reusability for humans. However, the features offered by PDF/UA also improve automated extraction of text. When tags are added correctly in a PDF/UA file, and when PDF readers process those tags correctly, all of the challenges mentioned above in section "Extracting Electronic Text – Document Segmentation/Tagging Issues" simply disappear (for guidance on implementing PDF/UA see the PDF/UA Technical Working Group's "Tagged PDF Best Practice Guide: Syntax").

A further reason for optimism is that Apple, Google, and Microsoft now support tagged documents in many of their applications (see "Apple Tags PDF" and "Microsoft Announces Forthcoming..."). While it still may be vexing to process many PDFs, there is hope that as these standards continue to take off and as the PDF Association and stakeholders continue to improve these standards and validation tools, PDF processing will become more straightforward, more secure, and more reliable.